UNIVERSITY OF GOTHENBURG

Evaluating the Haxe Programming Language

Performance comparison between Haxe and platform-specific languages

Bachelor of Science Thesis in Software Engineering and Management

STEPAN STEPASYUK
YAVOR PAUNOV

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2013

Evaluating the Haxe Programming Language
Performance comparison between Haxe and platform-specific languages

STEPAN STEPASYUK,
YAVOR PAUNOV.

Examiner: RICHARD TORKAR

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2013

Evaluating the Haxe Programming Language

Performance comparison between Haxe and platform-specific languages

Stepan Stepasyuk
Gothenburg University
Department of Computer Science and Engineering
Gothenburg, Sweden
stepan.stepasyuk@gmail.com

Yavor Paunov
Gothenburg University
Department of Computer Science and Engineering
Gothenburg, Sweden
yvr.paunov@gmail.com

Abstract

The goal of this paper is to evaluate the performance of Haxe, a cross-platform programming language. By applying an experimental method and a statistical test, we measure and analyze the performance associated with compiling Haxe source code to five of its targets. We find a significant difference in performance between Haxe-compiled and target-specific language code on four out of five targets, two of which in favour of Haxe and two against it. Our findings are useful for developers who are considering adopting the language with the intent of replacing their current development toolbox with it.

Keywords: *evaluation, performance, language design, experiment, statistical testing*

1. Introduction

The recent years have seen a growing demand for cross-platform development solutions[1], especially in the mobile sector where new platforms come out rather often and a big part the market is fragmented. In order to reach a wider audience, applications have to target multiple platforms and thus must be developed multiple times. This requires more effort from the developers and creates additional development and maintenance costs [1]. Tools that simplify this process already exist in great numbers and focus on different domains such as web, desktop and mobile.

One such tool is Haxe, a language with the ability to compile to a number of different platforms. Although not a complete cross-platform solution by itself, Haxe can greatly ease the process. For instance, platform-specific code still needs to be written but conditional compilation makes an abstraction layer easy to implement. Haxe can be useful when targeting a single platform, as well because many of its features are lacking on its targets

This paper aims to test how the performance of programs compiled from Haxe compares to code written in the platform-specific languages. This led to the following research question:

Is there a significant overhead related to compiling Haxe source code?

The targets used during the testing were Flash, C++, PHP, JavaScript, and Java. The architecture and the inner workings of the compiler, although of potential interest for a future study, are not within the scope of the paper.

2. Background

2.1 Haxe

Haxe[2] is a strictly typed object-oriented language whose standard libraries are licensed under the "two-clause" BSD license while the compiler is under GPLv2+ in order to "keep it free open software"[3]. Advertised mainly for its multiplatform nature Haxe compiles to a number of different programming platforms, including JavaScript, Flash, PHP, C++, and NekoVM and provides experimental support for C# and Java. The practical implications of this are that Haxe code can be compiled for applications running on desktop, mobile and web platforms. Haxe comes with a standard library that can be used on all supported targets and platform-specific libraries for each of them.

2.1.1 Multiplatform

The biggest selling point of Haxe is its multiplatform nature, a noticeable benefit of which is saving the trouble of switching languages. For example, in a web-development project where PHP is used for back-end and JavaScript for front-end. By using Haxe, it is possible to develop for both platforms using the same language.

Moreover, Haxe provides features missing in the languages of some of the target platforms. For instance, it provides *enumerations* that are missing from ActionScript 3 and PHP and *type safety* missing in loosely typed languages such as PHP and JavaScript.

2.1.2 Comprehensive type system

The Haxe type system comes with several different groups of types. For example, basic types like *class* or *enum*. The *class* type is similar to that in other object-oriented languages. It supports inheritance and can implement interfaces. The *enum* type which works in a similar way to that in Java.

Haxe also allows for more advanced types like the anonymous *structure* that represents a list of fields, which do not refer to a specific class. Structures can be made reusable by using the *typedef* keyword.

Haxe is a strictly typed language. However, there are several ways to get around compile-time type checking. One of them is by using the *untyped* keyword, which disables type checking for the specified block of code. Another option is to use the *Dynamic*

[1] http://software.intel.com/en-us/blogs/2013/03/07/cross-platform-development-what-the-stats-say [Accessed: 25 May 2013]

[2] http://haxe.org/doc/intro [Accessed: 25 May 2013]

[3] http://haxe.org/doc/license [Accessed: 25 May 2013]

type when declaring variables. A variable of the *Dynamic* type can hold a value of any other type. Moreover, an infinite number of fields can be added to a *Dynamic* variable. All of those will also be *Dynamic* unless a type parameter is given, in which case all fields will be of the specified type.

2.1.3 Code embedding

It is possible to use code written in some of the targeted languages while writing Haxe. This can be done with JavaScript, PHP, C# and Java. It is important to note that since C# and Java are experimental targets, code embedding for these targets is discouraged[4]. Code embedding can be achieved by utilizing Haxe *magic* functions. For example, the functions *__js__* and *__php__* will embed JavaScript and PHP code respectively. Using external libraries on the other targets is possible as well but not as straightforward and requires certain workarounds[5,6].

2.1.4 Compiler metadata

Haxe allows metadata annotation using the @ sign, in which it shares similarity to that of Java. What makes metadata in Haxe powerful is that under certain circumstances it can affect the way the compiler works. In order to mark compiler metadata, a colon character (':') is added in front of the identifier. A number of general identifiers can be used on all platforms in addition to a few platform-specific ones.

Examples of platform independent identifiers include the *@:final* annotation, preventing a class from being further extended. This is similar to the final keyword in ActionScript 3[7] and Java[8] , providing the same feature. Although, Haxe itself does not support method overloading, the *@:overload* identifier allows overloading of external methods. Other examples of compiler metadata are the *@:macro* and *@:build* identifiers, used to create macros.

2.1.5 Conditional compilation

Haxe offers the option of choosing which platforms to target with a given block of code. This can be of use when implementing a layer of platform abstraction.

3. Methodology

Experimental research [2, 3] was used for this study. More specifically, statistical hypothesis testing [4] was applied. First, null and alternative hypotheses were stated. After that, a simple performance benchmark was conducted on a number of platforms using a fractal-drawing algorithm. The data obtained from the benchmark was then analysed using a non-parametric two-tailed test to support or reject the null hypothesis.

The main motivation behind using an empirical approach was that it appeared to be the best method of evaluating performance. Moreover, traditional experimentation has been judged an integral part of computer science that practical application cannot replace [5].

3.1 Experimental setup

The experimental consisted of comparing the execution time of Haxe-compiled and platform-specific code on the following targets:

- Flash
- C++
- Java
- PHP
- JavaScript

The Mann–Whitney U test [6] was used in order to distinguish the presence of significant statistical difference between the results. This particular test was selected due to the nature of data to be compared. Samples of execution times are independent since execution time of one implementation does not affect the execution time of another in any way. Moreover, the obtained data is of a rational level[9] because it can be ranked (from shortest to longest time of execution and vice versa). It is meaningful because we can compare two given executions and find out which one was the quickest. It also has an absolute minimum of 0 seconds execution time, and it is possible to calculate ratios based on it. However, the absolute minimum is a subject for argument since no execution time can be exactly 0 seconds in practice. It can be 0.01 or 0.000001 and the amount of zeros between the decimal point and 1 can span infinitely. On the other hand, no algorithm can execute in less than 0 seconds, for instance -0.1 seconds.

The performed test was two-tailed which means that finding the presence of significant statistical difference and not its direction is the main goal of the test [6]. Thus, hypotheses were formulated accordingly. Null hypothesis was stating that overhead in performance of compiling from Haxe is statistically significant in comparison to compiling from platform-specific languages. Conversely, alternative hypothesis was stating that overhead in performance of compiling from Haxe is statistically insignificant in comparison to compiling from platform-specific code. The formal hypotheses of the experiment were defined as follows:

Null hypothesis (H_0):

$$O_{Haxe} \neq O_{PSL}$$

Alternative hypothesis (H_1):

$$O_{Haxe} = O_{PSL}$$

In those, PSL stands for platform-specific language.

An escape time algorithm used to generate the Mandelbrot fractal set was executed as part of the experiment. The algorithm takes three parameters as input:

- Width
- Height
- Maximum number of iterations

Width and height are the number of points in the set, on the horizontal and vertical axes respectively. Therefore, the total number of points is the product of the width and the height. The values used were 550 for the width and 400 for the height. Those values were chosen, as they were large enough to stress the CPU.

Since in some cases reaching the escape condition might take an extremely long time, we specify a maximum number of itera-

[4] http://haxe.org/doc/advanced/magic [Accessed: 12 June 2013]

[5] http://haxe.org/manual/swc [Accessed: 12 June 2013]

[6] http://haxe.org/doc/js/extern_libraries [Accessed: 12 June 2013]

[7] http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/statements.html [Accessed: 12 June 2013]

[8] http://docs.oracle.com/javase/tutorial/java/IandI/final.html [Accessed: 12 June 2013]

[9] http://infinity.cos.edu/faculty/woodbury/stats/tutorial/Data_Levels.htm [Accessed: 2 June 2013]

tions for each point in the set. More maximum iterations mean a more detailed representation of the set, but also a longer time to calculate it. The algorithm was run using 100 maximum iterations as, again, that number was judged high enough to stress the CPU.

Each implementation of the algorithm was executed 3 000 times, in order to produce sufficiently large samples [4] and alleviate errors. In total 10 implementations were used in the experiment. Among them are five Haxe-compiled and five compiled from platform-specific languages code. Common logging tools, available for each of the targets, were used for recording the execution times. First, Haxe-compiled implementation was run and its execution times were logged. After that, implementation for the same platform but compiled from platform-specific language code was run and its execution times were logged as well. These logs were used as samples to compare using the Mann-Whitney U test.

Two samples were ranked as one combined sample with respect to ranking ties specific for the Mann-Whitney U test. After that, the sums of ranks for both samples were calculated and used to determine the U values for both samples:

$$U_1 = n_1 n_2 + \frac{n_1 (n_1 + 1)}{2} - W_1$$

$$U_2 = n_1 n_2 + \frac{n_2 (n_2 + 1)}{2} - W_2$$

In the above formulas, n_1 and n_2 represent the count of sample 1 and 2 respectively, W_1 is the sum of ranks of sample 1 and W_2 is the sum of ranks of sample 2.

The same equation is to be applied when calculating U for sample 2.

Since large data samples were obtained during the experiment, z score was calculated in order to determine the probability of obtaining the observed result if H_0 is true.

$$\mu_U = \frac{n_1 n_2}{2}$$

$$\sigma_U = \sqrt{\frac{n_1 n_2 (n_1 + n_2 + 1)}{12}}$$

$$z = \frac{U - \mu}{\sigma}$$

Mann-Whitney U test requires using the lesser of U values to calculate the z score [6]. The p value is calculated based on the assumption that data is under normal distribution. If p is less than significance level, α which is 0.05 then H_0 is rejected and significant difference exists. Otherwise, H_0 is accepted.

Depending on whether a significant statistical difference was discovered between two implementations for the same platform, an effect size was measured in order to determine the strength of the relationship between given samples of execution times.

$$r = \frac{z}{\sqrt{N}}$$

In this formula, N is the total count of all samples. The absolute value of z is to be considered. If r lies in interval between 0.1 and 0.3 then the effect size is small. In case r is greater than 0.3 but less than 0.5 then the effect size is medium. Finally, if the value of r is greater than 0.5 then the effect size is large [7, 8].

3.1.1 Environment

The hardware used for testing had the following specifications:

- *CPU: Intel Celeron CPU 900 @ 2.2 GHz*
- *RAM: DDR2 2GB*
- *OS: Windows 7 Professional SP1 x86*

Haxe version 2.10 was used. For running JavaScript tests Mozilla Firefox 21 was used. Tests for PHP were conducted with the use of Apache Server 2.2.22 and PHP 5.4.3. Adobe Flex SDK 4.6.0 was used to compile the ActionScript 3 implementation with Adobe Flash Player 11 as the target. The stage parameters, both using Haxe and ActionScript 3 were 550 by 400, and 40 frames per second. The Microsoft 32-bit C/C++ Compiler 16.00 was used for compiling C++ code, both platform-specific and produced by Haxe. The JDK 1.7.0 was used to compile Java code. Obtained execution times were analysed using Microsoft Excel 2007.

3.1.2 Limitations

It is important to note that obtained results were affected by other applications running in the background while the tests were conducted [9]. Even though the number of background applications was minimized as much as possible, vital system processes were left working.

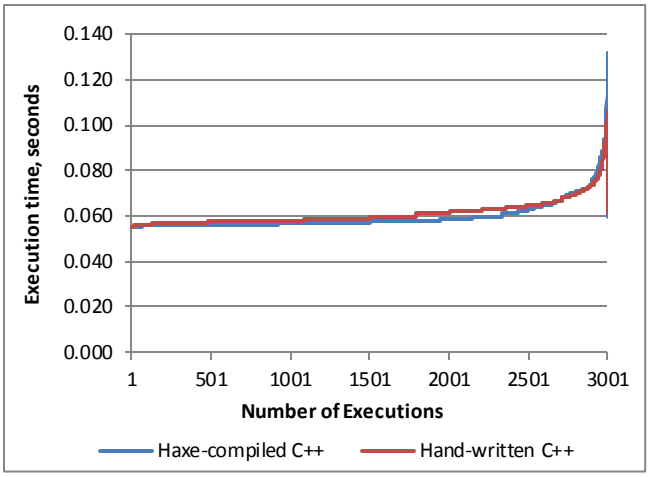
The written code was not optimized for each target individually. For instance, none of the authors of this study had any previous experience with C++, which in the end might have affected the outcome. If this experiment is to be recreated and the test algorithm is to be optimized for each target, the obtained results can differ from the ones that are found in this study.

4. Results

4.1 C++

The average execution time of a platform-specific implementation in C++ was 0.061 seconds while Haxe-compiled implementation was executed in an average of 0.060 seconds. However, the sum of ranks for Haxe-compiled implementation was 7,387,348 and the respective parameter for platform-specific implementation was 10,619,252. The p value was calculated to be 0.00, which was less than 0.05 supporting H_0. There was significant difference between the two implementations. Haxe-compiled implementation executed faster more times than platform-specific implementation, which resulted in the sum of ranks for Haxe-compiled code being less than respective parameter for platform-specific code. By following the formula for calculating the effect size r, the value of 0.31 was acquired meaning that the effect size was medium. Figure 1 demonstrates the comparison between the Haxe-compiled and the platform-specific implementations.

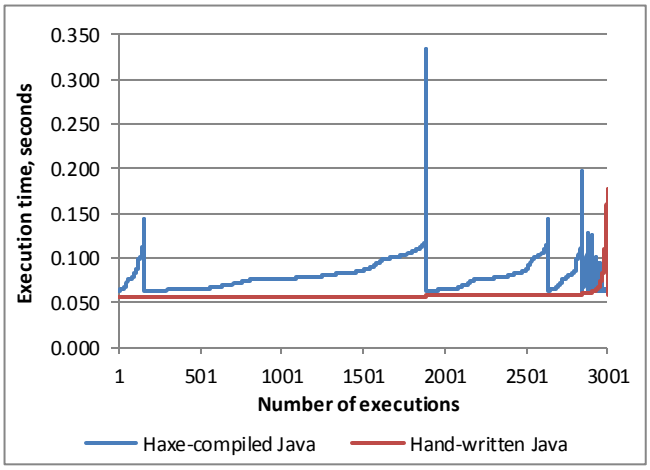
Figure 1. C++ execution times



4.2 Java

The average execution time of the platform-specific implementation in Java was estimated to be 0.058 seconds while the Haxe-compiled implementation was executed in an average of 0.079 seconds. The sum of ranks for the Haxe-compiled code was 13,375,332 and the same parameter for platform-specific code was 4,627,668. The p value was calculated to be 0.00, which was less than 0.05 meaning that H_0 stands. There was significant difference between the two implementations. The platform-specific implementation executed faster almost 3 times more often than its Haxe-compiled counterpart, which resulted in the sum of ranks for Haxe-compiled implementation being greater than respective parameter for platform-specific implementation. By following the formula for calculating the effect size r, the value of 0.84 was acquired meaning that the effect size was large. Figure 2 demonstrates the comparison between Haxe-compiled and platform-specific implementations.

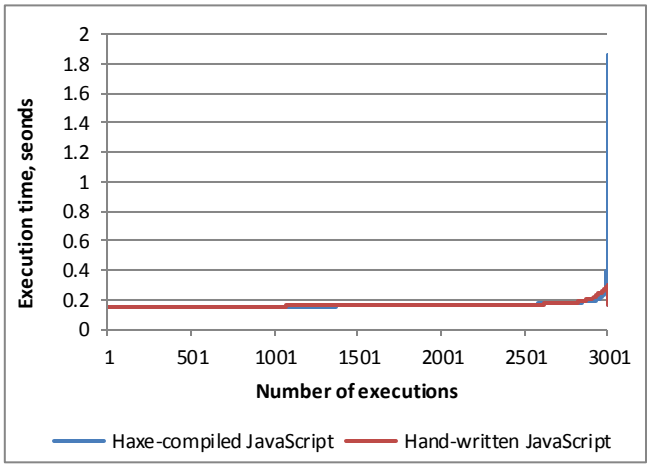
Figure 2. Java execution times



4.3 JavaScript

The average execution time of the platform-specific implementation in JavaScript was estimated to be 0.164 seconds while the Haxe-compiled implementation was executed in an average of 0.163 seconds. The sum of ranks for Haxe-compiled implementation was 8,669,517 and the respective parameter for the platform-specific implementation was 9,333,484. The p value was calculated to be 1.00 and was greater than 0.05 meaning that H_0 is rejected. There was no significant difference between the implementations. Haxe-compiled JavaScript code was as fast as the platform-specific code. Effect size was not calculated since no significant statistical difference was observed. The figure below demonstrates the comparison between Haxe-compiled and platform-specific implementations.

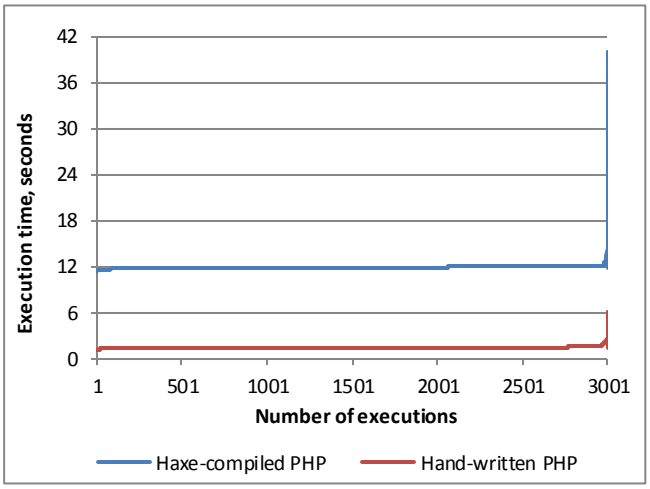
Figure 3. JavaScript execution times



4.4 PHP

The average execution time of the platform-specific code in PHP was estimated to be 1.405 seconds while Haxe-compiled code was executed in an average of 11.918 seconds. The sum of ranks for the Haxe-compiled implementation was 13,501,500 and the respective parameter for the platform-specific implementation was 4,501,500. The p value was calculated to be 0.00, which was less than 0.05 meaning that H_0 is supported. There was significant difference between the implementations. The sum of ranks for Haxe-compiled implementation was much greater than respective parameter for platform-specific implementation meaning significantly longer execution times. By following the above-mentioned formula for calculating the effect size, the value of 0.87 was acquired meaning that the effect size was large. The following figure demonstrates the comparison between Haxe-compiled and platform-specific implementations.

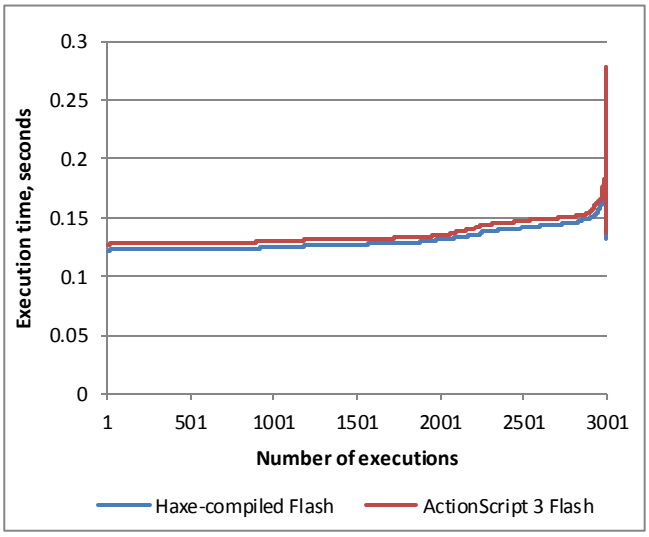
Figure 4. PHP execution times



4.5 Flash

The average execution time of the Flash implementation written in ActionScript 3 was estimated to be 0.136 seconds while the Haxe-compiled implementation was executed in average of 0.131 seconds. The sum of ranks for the Haxe implementation was 6,970,574 and the respective parameter for the ActionScript 3 implementation was 11,032,426. The p value was calculated to be 0.00 and was less than 0.05 meaning that H_0 stands. There was significant difference between the two implementations. The sum of ranks for Haxe was greater than respective parameter for ActionScript 3 implying longer execution times. By following the above-mentioned formula for calculating the effect size, the value of 0.39 was acquired meaning that the effect size was medium. The figure below demonstrates the comparison between Haxe-compiled and platform-specific implementations.

Figure 5. Flash execution times



5. Discussions

The only target where no significant overhead was found is JavaScript. With that in mind, and considering the advantage in terms of language design, Haxe is a feasible option for developing projects where performance is a priority.

The Flash target for Haxe turned out to be ahead of ActionScript 3 with a medium effect size in terms of execution time. The Haxe compiler performs translating to efficient byte code better than its ActionScript 3 counterpart, the Adobe Flex compiler. This can be beneficial in game development where the Flash Player is targeted extensively. A downside worth noting would be that although Haxe is similar to ActionScript 3, it is also different enough to make a new developer struggle with its peculiarities. However, the additional features such as the more comprehensive type system, macros and the ease of remoting with the backend can make up for that in the long term.

An unexpected result was found from testing the C++ target, which performed significantly better than platform-specific code. This, however, is at the expense of control over memory management, which is done using a garbage collector on the Haxe C++ target. Although, for an algorithm as small as the one used for the experiment, this is a non-issue. However, in case manual memory management is a requirement, then Haxe is not the right tool[10].

The PHP target had the greatest overhead. For that reason, it is difficult to recommend it for performance-critical applications. It should be noted here, however, that running an algorithm similar to the one we used on PHP does not represent a likely real-world scenario. PHP's strength lies within data-driven applications rather than calculations[11]. A more realistic scenario would include interaction with a database. In such cases, high maintainability is a priority [10].

Java code compiled by Haxe also performed significantly slower, which we attribute to the experimental status of the Java target at the time of writing of this paper. Currently, manually writing Java code remains the more viable option due to the performance benefits and a small support for Java API in Haxe[12].

Considering the performance on the JavaScript, Flash, and C++ targets, Haxe can be useful for developing applications where performance is of importance. However, for the majority of tasks one has to use each target's own library, which means that a large portion of the logic would have to be rewritten. To avoid that, one can use the NME framework, which allows using a multiplatform API mirroring the Flash API[13].

6. Conclusions

The conducted experiment indicated that Haxe is faster than C++ and ActionScript 3, slower than PHP and Java and runs almost as fast as JavaScript. Such performance and other features make it a viable alternative to platform-specific code in a number of projects where cross-platform abilities matter. It is relatively easy to start using the language especially if one has experience with ActionScript 3[14]. A number of community-created libraries and frameworks expand Haxe's functionality further and add more platforms to target.

7. Future Work

We reached interesting results, some of which were unexpected. For instance, the performance of the Haxe-produced C++ code was certainly a surprise. A study focusing on the causes behind such results could definitely build on this one. More specifically, a study on the architecture and inner workings of the compiler.

Second, a more extensive evaluation of Haxe in terms of its language design qualities would be an important contribution. Haxe is contains curious design concepts and has a set of features rarely found in a single language.

A practical approach would be studying the viability of developing a code converter for producing Haxe code from other languages. Such a tool could ease the porting of applications to Haxe by cutting time and costs required for such a procedure. The typical process of porting an application would involve re-writing and adapting its source code [11]. A conversion tool would greatly simplify this process by automating it. The community has already created experimental source code converters such as ActionScript 3 to Haxe and C# to Haxe.

---

[10] http://haxe.1354130.n2.nabble.com/How-mature-is-haXe-c-tp5076513p5079439.html [Accessed: 12 June 2013]

[11] http://www.php.net/manual/en/intro-whatcando.php [Accessed: 12 June 2013]

[12] http://haxe.org/doc/start/java [Accessed: 12 June 2013]

[13] http://haxe.org/forum/thread/3395#nabble-td5814135 [Accessed: 12 June 2013]

[14] http://www.grantmathews.com/43 [Accessed: 12 June 2013]

References

[1] J. Bishop and N. Horspool, “Cross-platform development: Software that lasts”, *Computer*, vol. 39, no. 10, pp. 26-35, 2006

[2] W. F. Tichy, P. Lukowicz, L. Prechelt and E. A. Heinz, "Experimental evaluation in computer science: A quantitative study", *Journal of Systems and Software*, vol. 28, no. 1, pp. 9-18, 1995.

[3] J. W. Creswell, Research Design: Qualitative, Quantitative, and Mixed Methods Approaches, 3rd ed., London: Sage Publications, 2009.

[4] A. Arcuri and L. Briand, "A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering", *Software Testing, Verification and Reliability*, 2012.

[5] W. F. Tichy, "Should computer scientists experiment more?", *Computer*, vol. 31, no. 5, pp. 32-40, 1998.

[6] K. Black, Business statistics: Contemporary Decision Making, 6th ed., 2010.

[7] R. G. Newcombe, Confidence Intervals for Proportions and Related Measures of Effect Size, vol. 51, CRC Press, 2012.

[8] V. B. Kampenes, T. Dybå, J. E. Hannay and D. I. Sjøberg, "A systematic review of effect size in software engineering experiments", *Information and Software Technology*, vol. 49, no. 11, pp. 1073-1086, 2007.

[9] F. I. Vokolos and E.J. Weyuker, Performance testing of software systems, *Proceedings of the 1st international workshop on Software and performance*, pp. 80-87, ACM, 1998.

[10] L. Lundberg, D. Häggander and W. Diestelkamp, “Conflicts and trade-offs between software performance and maintainability”, *Performance Engineering*, vol. 2047, pp. 56-67, Springer Berlin Heidelberg, 2001.

[11] Mindfire Solutions, *Porting: A Development Primer*, 2001